Phoebe Nichols

An Implementation of Prolog

Computer Science Tripos – Part II

Churchill College

Contents

Chapter 1 — Introduction

1.1 Project summary

TODO

1.2 Motivation

1.2.1 Declarative programming

Declarative programming languages are a family of programming language that describe logical properties of the result of a computation, but do not specify how the computation should be performed. This is in contrast to imperative programming languages, which describe a sequence of commands for the computer to perform. Logic programming languages are a class of declarative programming languages where the program consists of a series of logical statements describing the result of the program.

The motivation for logic programming is that it provides a useful layer of abstraction: the idea is that the programmer's life should be simplified if they need only be concerned with the logic associated with their program. is this achieved in practice? seem like it is not, prolog not popular Although logic programming languages are not the most popular why? , they have some important applications. Examples of these applications include:

- Datalog is a subset of Prolog that is used as a database query language. This language has modern applications in data integration, declarative networking, program analysis, information extraction, network monitoring, security, and cloud computing [1]. why is datalog good for this? For example, the Doop framework uses Datalog for points-to analysis of Java programs [2].
- Prolog has been used in the JVM 11 specification provide a formal description of semantics of the JVM [3].
- some standard AI application for prolog? hihi hi hi

- Motivation for project

1.3 Related work

The first logic programming language was called Baroque, and was implemented in 1972. This was an assembly-like language that used logical statements in the form of Horn clauses [4]. The first Prolog system was also implemented in 1972, by Alain Colmerauer and Philippe Roussel. They chose the name Prolog as an abbreviation for 'programmation en logique' (meanning programming in logic). Robert Kawalski and Maarten van Emden then formalised the use of predicate logic as a programming language [5],[6].

This early work on Prolog was followed by David Warren's invention of the Warren Abstract Machine [7]. This is an abstract machine for the efficient execution of Prolog, and is used by popular Prolog execution systems such as SWI-Prolog [8].

Mycroft, Mercury

Chapter 2 — Preparation

This chapter contains a discussion of the preparatory work done for each of the components of the project. It begins by discussing Prolog's features, execution model, and program structure. The Warren Abstract Machine is discussed—this is a common target for Prolog compilers—followed by an introduction to the Mycroft-OKeefe type system for Prolog. After the introduction of these concepts, preparatory work relating to the software engineering practice used is explained, including a discussion of testing strategy and development methodology. The chapter finishes with a quick summary of its key points.

2.1 Programming in Prolog

A Prolog program is a set of Horn clauses. Each rule is a horn clause: a disjunction of literals with at most one positive literal. A search is performed over these horn clauses to try to derive the given query.

2.1.1 Structure of a Prolog Program

A Prolog program is comprised of a set of predicates, followed by a query. Each predicate is made up of at least one clause, and each clause has a head containing a list of argument terms and a body containing a list of calls to predicates. These concepts are described in more detail below.

Terms Prolog's only data structure is the term. A term is either an unbound variable, or a compound term consisting of a name and a (possibly empty) list of terms.

Goals A goal consists of a predicate name and a list of argument terms to the predicate. When Prolog executes a goal, it searches for an interpretation of the argument terms that satisfies the predicate.

Clauses Clauses represent logical statements that are true in the world described by the Prolog program. Each clause is a fact or a rule:

- A fact consists of a predicate name and a list of argument terms; the meaning of a fact is that it defines the given instance of the relation as being true.
- A rule consists of a head and a body. The head is a predicate name and a list of argument terms. The body is a non-empty list of goals. The meaning of a rule is that if we can show all the goals in the body are true then we can derive that the head is true.

Predicates A predicate is defined by the set of clauses with the same name and arity. A goal for a predicate can be derived from any of the clauses of the predicate, but the clauses are tried in the order that they are found in the program.

Queries A query is a list of goals that are to be derived by the Prolog program, and a Prolog program has at most one query. The execution of a program is a search for a substitution to the variables in the query that gives a satisfying interpretation for all the goals in the query; the result of the execution is either such an interpretation, failure, or an infinite loop.

2.2 The execution mechanism of Prolog

what order of logic is prolog

simple summary

Somewhere: clause precidence!

depth first search

2.2.1 Linear resolution

Resolution is an inference rule that combines two complementary literals. A statement of resolution in prepositional logic is

$$\frac{\neg A \to B \qquad B \to C}{\neg A \to C} \tag{2.1}$$

Linear resolution is a technique for first order logic theorem proving where resolution is repeatedly applied, with each resolvent being used as a parent clause for the next resolutions step.

give substitution rule

The basic execution mechanism of Prolog uses SLD (Simple Linear Definite Clause) resolution; a definite clause is a Horn clause with at most one positive literal. This is a special form of linear resolution for Horn clauses based on Kuehner and Kowalski's SL-resolution [9]—this performs linear resolution using a selection function to choose the literal to be resolved. In the case of SLD resolution, one of the literals to be resolved is the head of a clause and the other is a member of the body of a clause.

2.2.2 Unification

Unification is the operation of finding a minimal? substitution that gives the same resultant term when applied to a pair of possibly-distinct terms (A, B). A substitution is a mapping from the variables in a term to terms that replace these variables when the substitution is applied. In order for a goal to be derived using the resolution rule given in equation 2.1 the goal must be unified with the head of some clause. It is through this process of unification that Prolog variables are instantiated.

Unification is the basic operation of Prolog. When a clause `clause1(t1,...,tn) :- body.` is called with arguments `a1,...,an`, each `ti` is unified with `ai` before the body calls are performed. Here, 'calling' a clause corresponds to performing resolution with the clause. The individual Prolog terms unify as follows:

- The unification of a variable with any other term succeeds and binds the value of the variable to the value of the other term.
- Compound terms unify if their top function symbol and arity are the same and their lists of arguments unify recursively. It should be noted that these lists of argument terms might be empty.

2.2.3 Search strategy

Prolog performs a depth-first, left-to-right search for the solution to a query: when trying to derive a list of goals, the first goal must be derived before Prolog progresses to the second goal. It is important that a logic programming language specify its search strategy because the same program may give a different ordering of results when it is executed with various search strategies example program?. There are advantages to other search strategies over depth first search: breadth first search will always find a solution if one exists, whereas depth first search may get stuck in a loop.

2.2.4 Cut

Cut, written `!`, is a predicate that may be found in the body of a clause. The effect of a cut is to:

1. Commit to the current clause, meaning that no other clause from the same predicate will be backtracked to within this call to the predicate.
2. And discard all choice points created by executing goals inside the current clause up until the cut.

Cut is known as an extra-logical predicate because it does not have a declarative reading. Cut is, however, useful for improving the efficiency of Prolog programs since cut can prevent unnecessary backtracking. Cut also increases the expressivity of Prolog, for example cut can be used to implement logical negation.

2.2.5 Negation

It was relevant to understand negation in Prolog to understand if this was a feature that I should implement. Negation can be implemented using cut and so I decided that it was not a core feature for a Prolog compiler and therefore did not choose to implement it.

In Prolog, cut provides negation . This is demonstrated by the following example of using cut to define a `not_unify` clause:

```
unify(X,X).

not_unify(A,B) :- unify(A,B), !, fail.
not_unify(A,B).
```

This program uses the `fail` predicate: this is simply a predicate that always fails to be derived.

2.3 The Warren Abstract Machine

The Warren Abstract Machine (WAM) was designed by David Warren in 1983 [7] and has become the standard target for Prolog compilers. It defines an instruction set that can be efficiently interpreted, and serves as a useful intermediate representation when compiling Prolog to native code.

2.3.1 Variable representation

An important feature of the WAM is that it represents variables as pointers. The unification of a variable with a structure is therefore implemented by making the variable point to the representation of the structure in memory. Similarly, the unification of two variables is represented

by making one variable point to the other. Multiple applications of unification can lead to the creation of chains of pointers.

wam variable binding higher lower trailing rule?

link to discussion of aliasing problem

2.3.2 Instructions

The WAM instruction set contains five main types of instruction:

1. *Get* instructions, used to fetch a procedure's arguments from a standard location. where do I define procedure?
2. *Put* instructions, used to load the arguments to a procedure into a standard location.
3. *Unify* instructions, used to perform unification. These instructions verify that two terms have the same structure, and unify variables within the terms to enforce this property.
4. *Procedural* instructions, used for control transfer and environment allocation for procedure calls.
5. *Indexing* instructions, used to link the different clauses making up a procedure.

These instructions are interpreted by the WAM, in the same way as Java byte-code instructions are interpreted by the JVM.

2.3.3 Memory areas

The WAM has three main areas of memory: the local stack, heap, and trail stack.

1. The *local stack* contains environments and choice points. Environments contain the state associated with the call to a predicate: they are similar to stack frames on traditional architectures. Choice points contain the information needed to backtrack to an earlier point in the computation.
2. The *heap* contains structures and lists created by unification; these are pointed to by variables in the stack.
3. The *trail stack* contains references to variables that were bound on unification. These references are stored so that they can be unbound on backtracking.

+ detail

2.4 The Mycroft-OKeefe type system

The Mycroft-OKeefe type system [10] is a polymorphic type system for Prolog, based on the Hindley-Milner type system. Type definitions in this type system consist of a type name–possibly associated with a list of type variables–followed by a list of different type constructors for this type, possibly each using the type variables. Type declarations are needed for each predicate, and the types of variables are inferred. Figure 2.1 compares an implementation of append in ML to an implementation using the Mycroft-OKeefe type sytem.

```
datatype 'a list = Cons of 'a * 'a list | Nil

fun append(x,Nil) = Cons(x,Nil)
| append(x,Cons(y,z)) = Cons(y,append(x,z))
```

(A) *Append program in ML*

```
type list A = cons(A,list(A)), nil.

pred append(B, list(B), list(B)).
append(X,nil,cons(X,nil)).
append(X,cons(Y,Z),cons(Y,Res)) :- append(X,Z,Res).
```

(B) *Append program in Prolog using the Mycroft-OKeefe type system*

LISTING 2.1: *Comparison between an append function implemented in ML and using the Mycroft-OKeefe type system*

2.4.1 Type rules?

2.5 Evaluation strategy

2.6 Choice of tools

2.6.1 OCaml

I chose to implement my project in OCaml. There are a number of reasons for this decision:

Lexer and parser generator tools OCaml has good implementations of the standard lexer and parser generator tools `lex` and `yacc`, called `Ocamllex` [11] and `Menhir` [12]. It was useful to use these tools instead of implementing a lexer and parser from scratch, particularly because the grammar was modified throughout the development of the project.

Library support There is good library support for OCaml, both from its standard library library and from external libraries such as the Jane Street core library. For this project, the Jane Street core library was used extensively to provide performant implementations of standard data-types.

Static type checking and parametric polymorphism OCaml's type system facilitates rapid development thanks to its type system. Parametric polymorphism means functions work across multiple data-types without any extra effort, and static type checking allows many program errors to be detected at compile time.

Garbage collection Having a garbage collector is useful in general because the programmer need not be concerned with memory management. Garbage collection is particularly useful for this project because the abstract machine implementation can use OCaml's garbage collector implicitly eval this? talk about why we want to garbage collect prolog?

Pattern matching OCaml includes pattern matching syntax: this syntax has been used throughout the project, from walking the parse tree to driving the abstract machine. The project code would have taken longer to write and been more difficult to understand without this useful syntax.

Familiarity with ML I am familiar with ML from the IA course 'Foundations of Computer Science'. OCaml is based on ML and has many similar features, so I have been able to

learn the language quickly.

2.6.2 Lexer and parser generators

As mentioned above, the lexer and parser generators `Ocamllex` and `Menhir` were used. There was no alternative to `Ocamllex`, but an alternative to `Menhir` is `Ocamllyac` [11]. `Menhir` was chosen because it has the advantage of generating more comprehensible error methods [12].

2.6.3 OUnit

The OCaml unit test framework `OUnit` [13] was used to write tests for the project. This was chosen because it is simple and well-documented; I could find example code matching my use-cases easily.

2.7 Starting point

The starting point given in my project proposal is copied below.

> **Prolog as a programming language**
>
> I will need a good understanding of how to program in Prolog in order to implement the Prolog compiler. I will also need to write test programs for the subset of Prolog that I have chosen to compile. These tests will help me to evaluate the correctness and performance of my compiler and abstract machine implementation.
>
> I am studying the 50% course, which does not cover Prolog until Lent term this year. I have, therefore, studied the Prolog course over the summer.
>
> **Prolog compilation and execution**
>
> I have no prior experience with implementing Prolog interpreters or compilers. The IB 75%/II 50% Prolog course that I studied over the summer introduced me to the basics of Prolog execution. I have also read around this area over the summer: some of the papers I read are cited in this document. I have not previously implemented any part of a Prolog compiler, interpreter, or abstract machine.
>
> **OCaml**
>
> I am intending to implement all of my project in OCaml initially, although I may add some SWI-Prolog for extension features. I have chosen OCaml because I have experience implementing lexers and parsers in ML from the compilers course, and found ML convenient for this sort of thing. OCaml will have the same advantages as ML, but is more widely used and therefore has more tools and libraries available.
>
> I have very little prior experience with OCaml (only from the IB compilers course), so I will need to dedicate time to studying it. This time is provisioned in my project plan.

2.8 Requirements analysis

It is important to define the requirements of a project, both so that crucial features can be focussed on and to provide a means of assessing whether the project has been a success. The

core requirements for my project is to implement the following components for a basic subset of Prolog:

- A lexer and parser should be implemented, either manually or using tools to generate these automatically.
- A translator from the parse tree to byte-code should be implemented.
- Abstract machine to execute this byte-code should be implemented. This abstract machine is indented to use a similar instruction set to the WAM, but does not need to implement all of its features.

There is no requirement concerning the performance of this system, but its performance should be evaluated and discussed in the evaluation section.

2.9 Software engineering techniques

- What is your software engineering technique?

2.9.1 Testing strategy

- Describe word undertaken before code written

- Show how project proposal was refined and clarified

2.10 Summary

A summary of the key points in this chapter is given below.

- A Prolog program consists of a set of clauses followed by a query. Each clause is a statement in first order logic: writing a clause in a Prolog program is like saying to the Prolog interpreter 'this statement holds'. Prolog performs a depth-first search over the clauses of a program to look for a satisfying interpretation for a given query; the result of a Prolog program is either a satisfying interpretation to the given query, failure, or an infinite loop.
- The Warren Abstract Machine is a common target for Prolog compilers. It defines abstract instruction set for Prolog that can be efficiently interpreted, similarly to how the JVM interprets Java byte-code.
- Prolog can be typed using the Mycroft-OKeefe type system; this is a polymorphic type system similar to that of ML.
- The project was implemented in OCaml, using automated lexer and parser generator tools. This decision was made because OCaml has many desirable properties for this project, including: these good lexer and parser generator tools, static type checking, parametric polymorphism, pattern matching, good library support, garbage collection, and my prior familiarity with ML.
- A blahblah development methodology was used to blahblah

Chapter 3 — Implementation

3.1 Overall structure

The overall pipeline of my project is given in figure 3.1.

3.2 Lexer and parser

The lexer and parser were implemented using the tools `ocamllex` [11] and `menhir` [12]. useful for iterative development methodology

3.3 Initial interpreter

3.3.1 Variable representation

3.4 Abstract machine

3.4.1 Variable representation

3.5 Code generation

3.5.1 The instruction set

- Actual instruction set?

3.6 Code optimisation

3.6.1 Last call optimisation

3.6.2 Using type information

3.7 Code execution

- Data structures used?

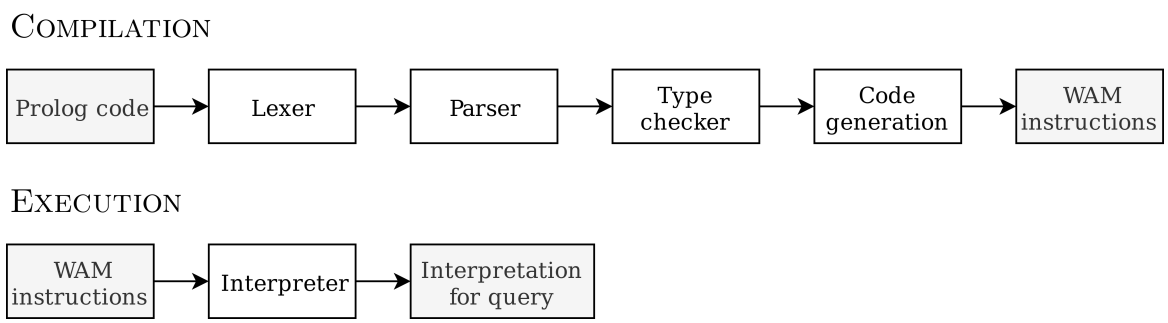

Figure 3.1: *Project pipeline*

3.7.1 Stack representation

3.8 Design strategy

- Design strategy (does it look ahead to testing)

3.9 Repository overview

- Repository overview - describe high level structure of your source code

---

- Describe what was produced

- Aliasing problem somewhere

3.10 Summary

Chapter 4 — Evaluation

Try to explain the results instead of just stating them

4.1 Success criterion

- Show project was a success

- How many of the original goals were achieved? -> were they proved to have been achieved

4.2 Compiler correctness

- Sample output

4.3 Speed

- Compare to other prologs

- How effective were optimisations?

4.4 Memory use

- Compare to other prologs

4.4.1 Last call optimisation

4.5 Performance gains from optimisation

4.6 Parallelisation?

4.7 How to improve performance

Where does most memory go?

Where does most time go?

- Show an ambitious case

- Tables of benchmark programs

- Graphs need confidence levels

4.8 Summary

Chapter 5 — Conclusions

- Short
- Summary
- Text-

Bibliography

[1] Todd Jeffrey Green Shan Shan Huang and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. *SIGMOD*, pages 1213 – 1216, 2011.

[2] Y. Smaragdakis and M. Bravenboer. Using datalog for fast and easy program analysis. *Datalog 2.0 Workshop*, 2010.

[3] Gilad Bracha Alex Buckley Daniel Smith Tim Lindholm, Frank Yellin. The java® virtual machine specification java se 11 edition, 2018.

[4] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 2002.

[5] Robert Kowalski. Predicate logic as programming language. volume 74, pages 569–574, 01 1974.

[6] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733 – 742, 1976.

[7] David H. D. Warren. An abstract Prolog instruction set. *SRI international*, 1983.

[8] Jan Wielemaker. Swi-prolog 6.1. `www.swi-prolog.org/download/devel/doc/SWI-Prolog-6.1.5.pdf`. Reference Manual.

[9] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 1971.

[10] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23(3):295 – 307, 1984.

[11] Lexer and parser generators (ocamllex, ocamlyacc). `https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`. OCaml Documentation.

[12] Jason Hickey Anil Madhavapeddy and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media.

[13] OUnit. `hhttp://ounit.forge.ocamlcore.org/`. OCaml unit testing framework.